下载APP

20 | 图像分割（下）：如何构建一个图像分割模型？

2021-11-26 方远

《PyTorch深度学习实战》 课程介绍 >

讲述：方远
时长 12:37　大小 11.56M

你好，我是方远。

在上一节课中，我们掌握了图像分割的理论知识，你是不是已经迫不及待要上手体验一下，找找手感了呢？

今天我们就从头开始，来完成一个图像分割项目。项目的内容是，对图片中的小猫进行语义分割。为了实现这个项目，我会引入一个简单但实用的网络结构：UNet。通过这节课的学习，你不但能再次体验一下完整机器学习的模型实现过程，还能实际训练一个语义分割模型。

数据部分

我们还是从机器学习开发三件套：数据、训练、评估说起。首先是数据准备部分，我们先对训练数据进行标记，然后完成数据读取工作。

分割图像的标记

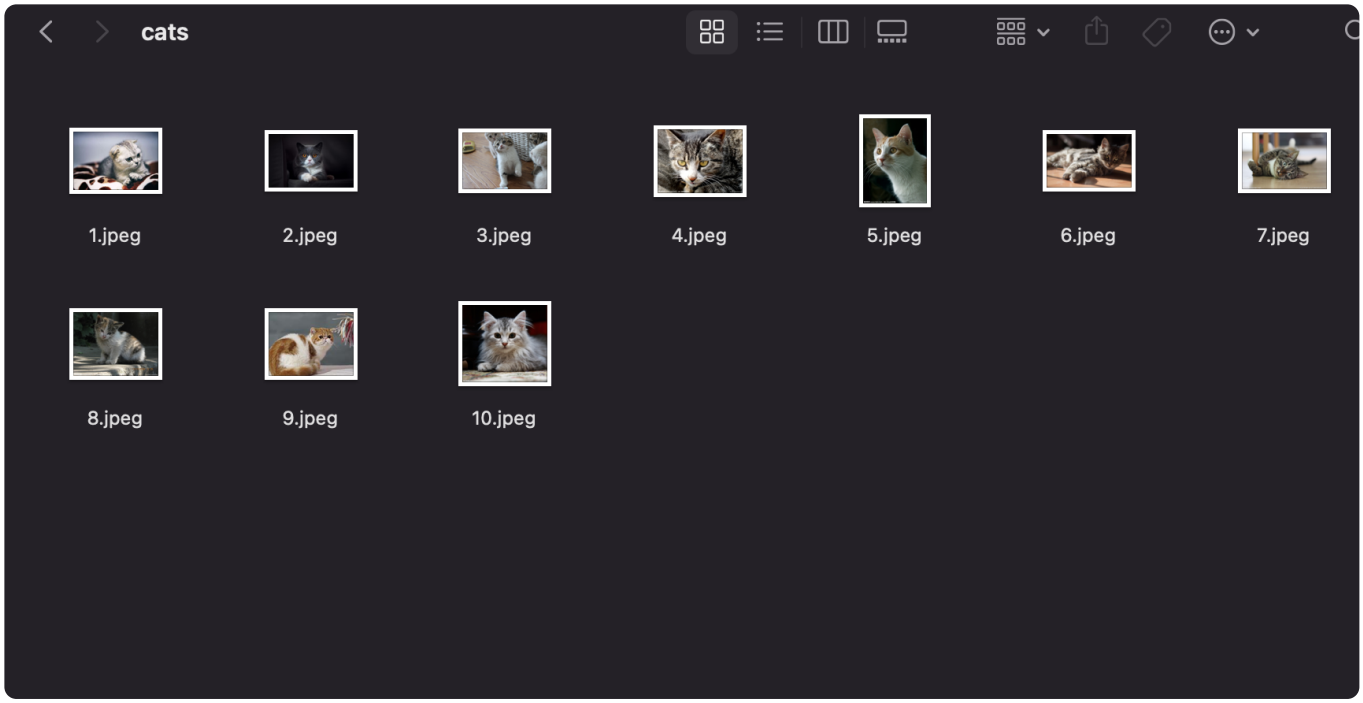
之前也提到过，图像分割的准备相比图像分类的准备更加复杂。那我们如何标记语义分割所需要的图片呢？在图像分割中，我们使用的每张图片都要有一张与之对应的 Mask，如下所示：

上节课我们说过，Mask 就是含有像素类别的特征图。结合这里的示例图片，我们可以看到，Mask 就是原图所对应的一张图片，它的每个位置都记录着原图每个位置对应的像素类别。对于 Mask 的标记，我们需要使用到 Labelme 工具。

标记的方法一共包括七步，我们挨个看一下。

第一步，下载安装Labelme。我们按照 Github 中的安装方式进行安装即可。如果安装比较慢的话，你可以使用国内的镜像（例如清华的）进行安装。

第二步，我们要将需要标记的图片放到一个文件夹中。这里我是将所有猫的图片放入到 cats 文件夹中了。



第三步，我们事先准备好一个 label.txt 的文件，里面每一行写好的需要标记的类别。我的 label.txt 如下：

```
__ignore__
_background_
cat
```

这里我要提醒你的是，前两行最好这么写。不这样写的话，使用 label2voc.py 转换就会报错，但 label2voc.py 不是唯一的数据转换方式（还可以使用 labelme_json_to_dataset，但推荐你使用 label2voc.py）。从第三行开始，表示要标记的类别。

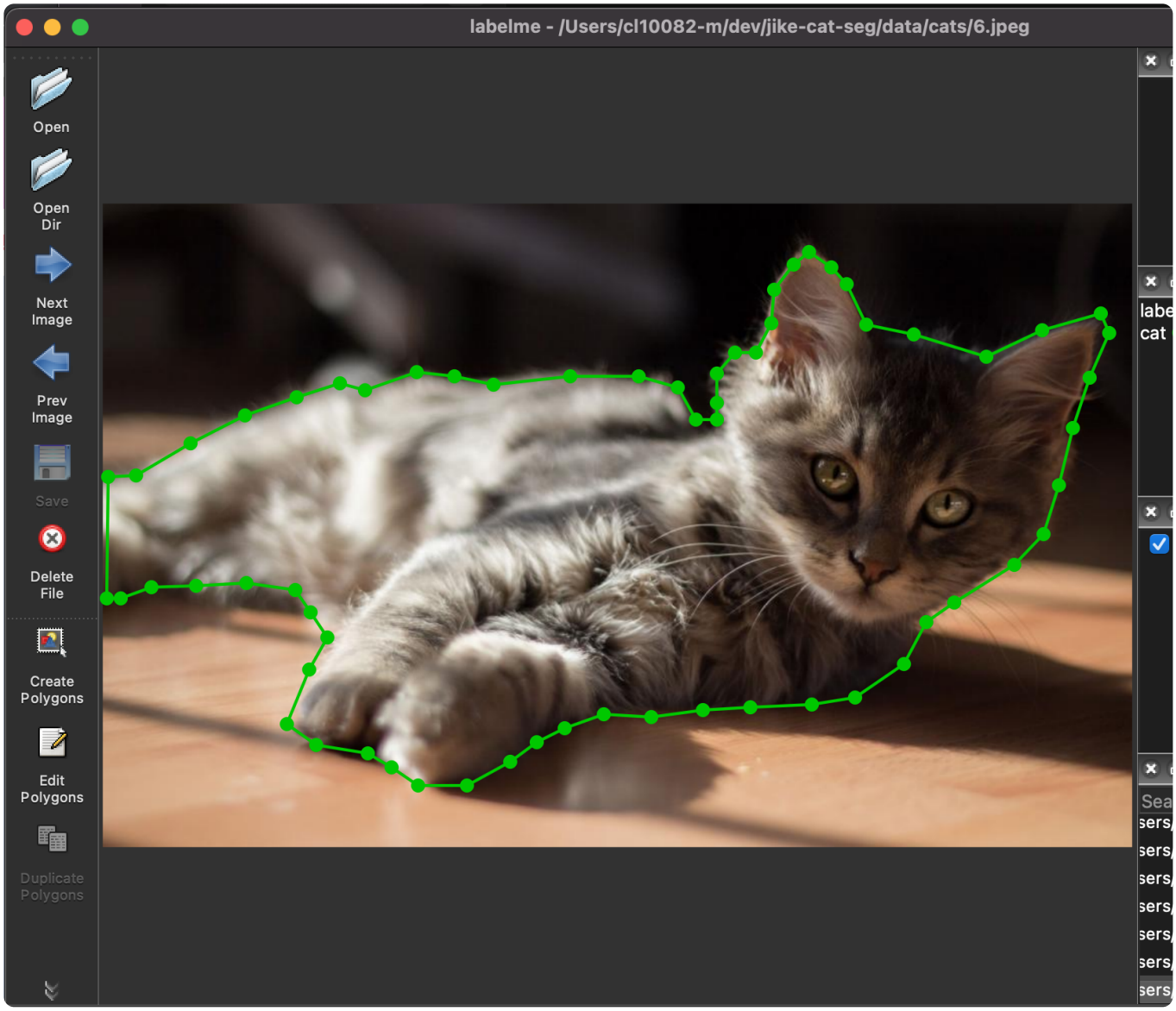
第四步，执行后面的这条命令，就会自动启动 Labelme。

```
labelme --labels labels.txt --nodata
```

第五步，点我们击左侧的 Open Dir，选择第二步中的文件夹，就会自动导入需要标记的图片。在右下角选择需要标记的文件后，会自动显示出来，如下图所示。

第六步：点击左侧的 Create Polygons。就可以开始标注了。标记的方式就是将小猫沿着它的边界给圈出来，当形成一个闭环的时候，Labelme 会自动提示你输入类别，我们选择 cat 类即可。

标记成功后，结果如下图所示。



当标记完成后，我们需要保存一下，保存之后会生成标记好的 json 文件。如下所示：

复制代码

```
fangyuan@geektime data $ ls cats
1.jpeg  1.json  10.jpeg 10.json 2.jpeg  3.jpeg  4.jpeg  4.json
```

第七步，执行下面的代码，将标记好的数据转换成 Mask。

复制代码

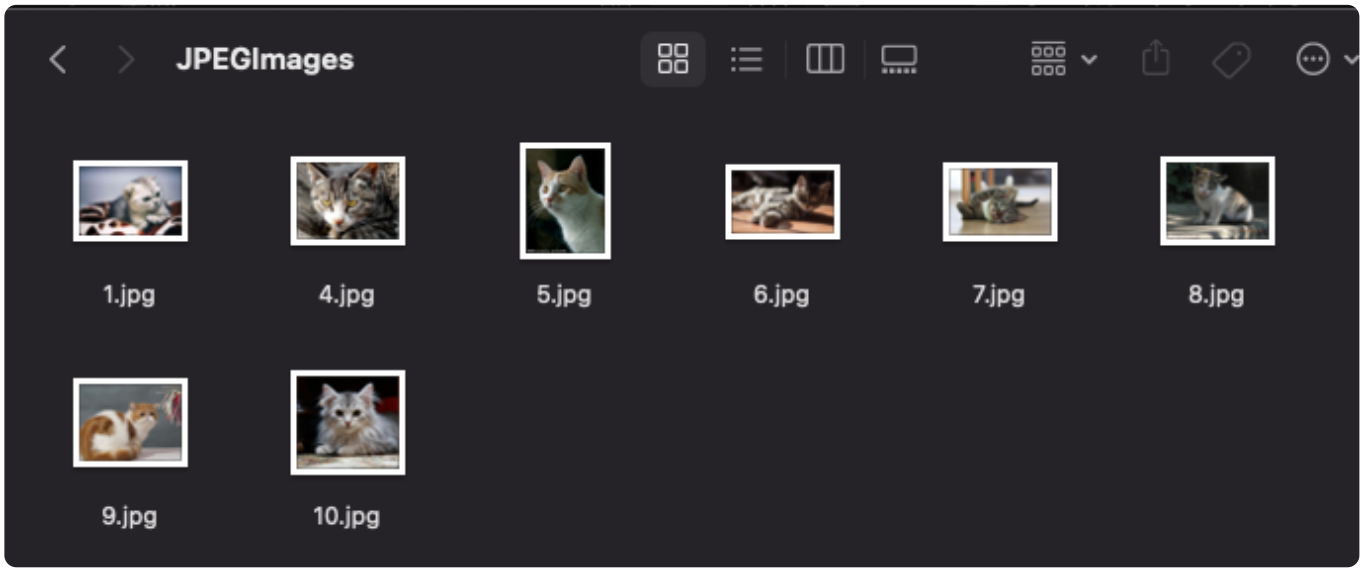
```
python label2voc.py cats cats_output --label label.txt
```

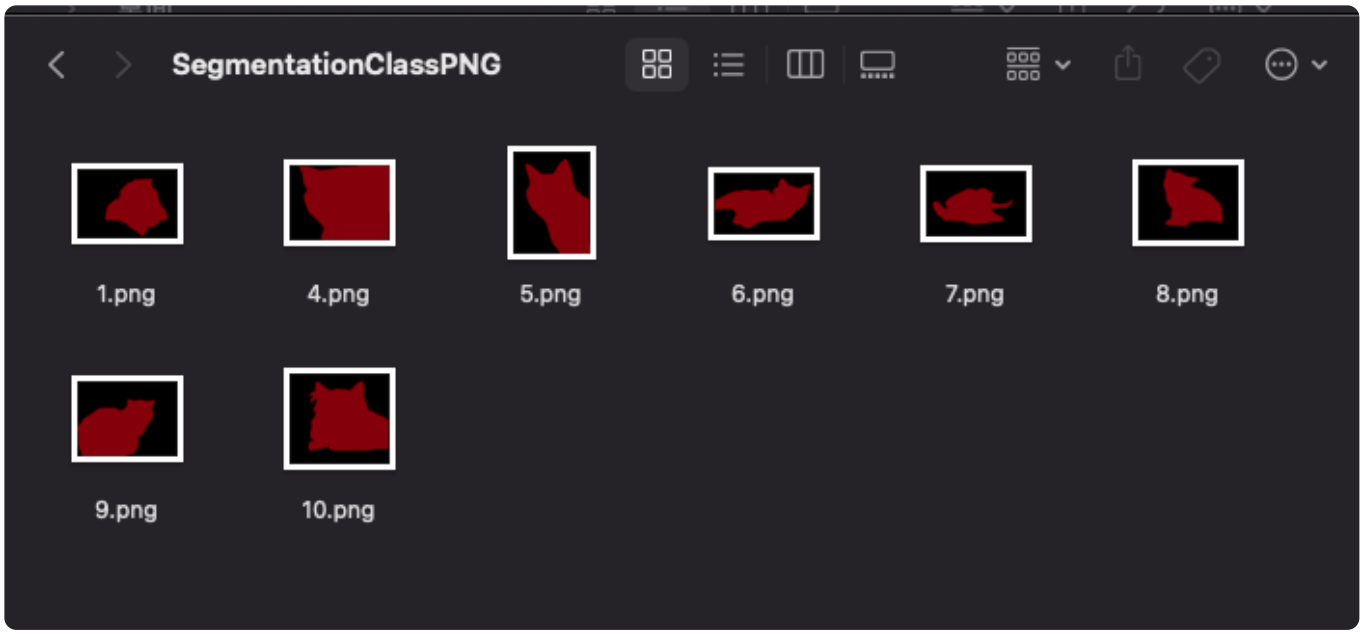
上面代码里用到的 label2voc.py，你可以通过后面这个链接获取它：
https://github.com/wkentaro/labelme/blob/main/examples/semantic_segmentat

ion/labelme2voc.py。

其中，cats 为标记好的数据，cats_output 为输出文件夹。在 cats_output 下会自动生成 4 个文件夹，我们只需要两个文件夹，分别是 JPEGImages（训练原图）与 SegmentationClassPNG（转换后的 Mask）。

到此为止，我们的数据就准备好了。我一共标记了 8 张图片，如下所示。当然了，在实际的项目中需要大量标记好的图片，这里主要是为了方便演示。





到此为止，标记工作宣告完成。

数据读取

完成了标记工作之后，我们就要用 PyTorch 把这些数据给读入进来了，我们把数据相关的写在 dataset.py 中。具体还是和之前讲的一样，要继承 Dataset 类，然后实现 __init__、__len__ 和 __getitem__ 方法。

dataset.py 的代码如下所示，我已经在代码中写好注释了，相信结合注释你很容易就能领会意思。

复制代码

```
import os
import torch
import numpy as np

from torch.utils.data import Dataset
from PIL import Image



class CatSegmentationDataset(Dataset):

    # 模型输入是3通道数据
    in_channels = 3
    # 模型输出是1通道数据
    out_channels = 1

    def __init__(
        self,
        images_dir,
        transform=None,
        image_size=256,
    ):

        print("Reading images...")
        # 原图所在的位置
        image_root_path = images_dir + os.sep + 'JPEGImages'
        # Mask所在的位置
        mask_root_path = images_dir + os.sep + 'SegmentationClassPNG'
        # 将图片与Mask读入后，分别存在image_slices与mask_slices中
        self.image_slices = []
        self.mask_slices = []
        for im_name in os.listdir(image_root_path):
            # 原图与mask的名字是相同的，只不过是后缀不一样
            mask_name = im_name.split('.')[0] + '.png'

            image_path = image_root_path + os.sep + im_name
            mask_path = mask_root_path + os.sep + mask_name

            im = np.asarray(Image.open(image_path).resize((image_size, image_s
            mask = np.asarray(Image.open(mask_path).resize((image_size, image_
```

```
            self.image_slices.append(im / 255.)
            self.mask_slices.append(mask)

        self.transform = transform

    def __len__(self):
        return len(self.image_slices)

    def __getitem__(self, idx):

        image = self.image_slices[idx]
        mask = self.mask_slices[idx]

        # tensor的顺序是（Batch_size，通道，高，宽）而numpy读入后的顺序是(高，宽，通道)
        image = image.transpose(2, 0, 1)
        # Mask是单通道数据，所以要再加一个维度
        mask = mask[np.newaxis, :, :]

        image = image.astype(np.float32)
        mask = mask.astype(np.float32)

        return image, mask

```

然后，我们的训练代码写在 train.py 中，train.py 中的 main 函数为主函数，在 main 中，我们会调用 data_loaders 来加载数据。代码如下所示：

复制代码

```
import torch

from torch.utils.data import DataLoader
from torch.utils.data import DataLoader
from dataset import CatSegmentationDataset as Dataset

def data_loaders(args):
    dataset_train = Dataset(
        images_dir=args.images,
        image_size=args.image_size,
        transform=transforms.ToTensor(),
    )

    loader_train = DataLoader(
        dataset_train,
        batch_size=args.batch_size,
        shuffle=True,
        num_workers=args.workers,
    )

```

```
    return loader_train

# args是传入的参数
def main(args):
    loader_train = data_loaders(args)
```

以上就是数据处理的全部内容了。接下来，我们再来看看模型训练部分的内容。

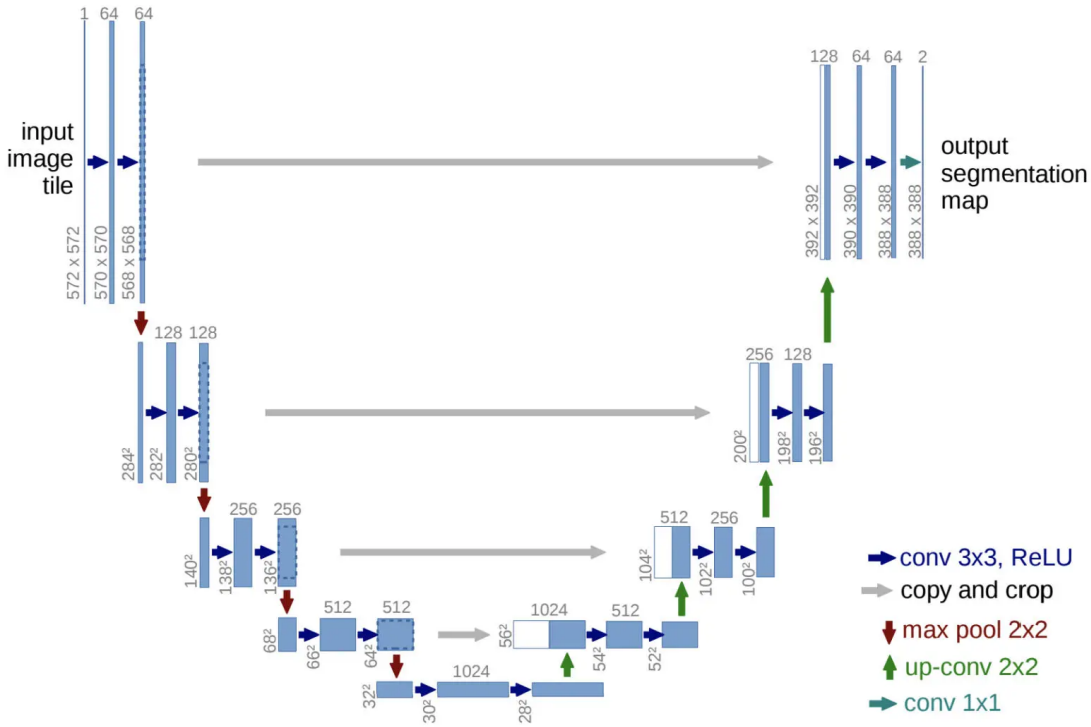
模型训练

我们先来回忆一下，模型训练的老三样，分别是网络结构、损失函数和优化方法。

先从网络结构说起，今天我要为你介绍一个叫做 UNet 的语义分割网络。

网络结构：UNet

UNet是一个非常实用的网络。它是一个典型的 Encoder-Decoder 类型的分割网络，网络结构非常简单，如下图所示。



图片来自论文：https://arxiv.org/pdf/1505.04597.pdf

它的网络结构虽然简单，但是效果并不“简单”，我在很多项目中都用它与一些主流的语义分割做对比，而 UNet 都取得了非常好的效果。

整体网络结构跟论文中给出的示意图一样，我们重点去关注几个实现细节。

第一点，图中横向蓝色的箭头，它们都是重复的相同结构，都是由两个 3x3 的卷积层组合而成的，在每层卷积之后会跟随一个 BN 层与 ReLU 的激活层。按照第 14 节课讲的，这一部分重复的组织是可以单独提取出来的。我们先来创建一个 unet.py 文件，用来定义网络结构。

现在 unet.py 中创建 Block 类，它是用来定义刚才所说的重复的卷积块：

复制代码

```
class Block(nn.Module):

    def __init__(self, in_channels, features):
        super(Block, self).__init__()

        self.features = features
        self.conv1 = nn.Conv2d(
                            in_channels=in_channels,
                            out_channels=features,
                            kernel_size=3,
                            padding='same',
                        )
        self.conv2 = nn.Conv2d(
                            in_channels=features,
                            out_channels=features,
                            kernel_size=3,
                            padding='same',
                        )

    def forward(self, input):
        x = self.conv1(input)
        x = nn.BatchNorm2d(num_features=self.features)(x)
        x = nn.ReLU(inplace=True)(x)
        x = self.conv2(x)
        x = nn.BatchNorm2d(num_features=self.features)(x)
        x = nn.ReLU(inplace=True)(x)

        return x
```

这里需要注意的是，同一个块内，特征图的尺寸是不变的，所以 padding 为 same。

第二点，就是绿色向上的箭头，也就是上采样的过程。这块的实现就是采用上一节课所讲的转置卷积来实现的。

最后一点，我们现在是要对小猫进行分割，也就是说一共有两个类别——猫与背景。对于二分类的问题，我们可以直接输出一张特征图，然后通过概率来进行判断是正例（猫）还是负例（背景），也就是下面代码中的第 71 行。同时，下述代码也补全了 unet.py 中的所有代码。

复制代码

```
import torch
import torch.nn as nn

class Block(nn.Module):
    ...
class UNet(nn.Module):

    def __init__(self, in_channels=3, out_channels=1, init_features=32):
        super(UNet, self).__init__()

        features = init_features
        self.conv_encoder_1 = Block(in_channels, features)
        self.conv_encoder_2 = Block(features, features * 2)
        self.conv_encoder_3 = Block(features * 2, features * 4)
        self.conv_encoder_4 = Block(features * 4, features * 8)

        self.bottleneck = Block(features * 8, features * 16)

        self.upconv4 = nn.ConvTranspose2d(
            features * 16, features * 8, kernel_size=2, stride=2
        )
        self.conv_decoder_4 = Block((features * 8) * 2, features * 8)
        self.upconv3 = nn.ConvTranspose2d(
            features * 8, features * 4, kernel_size=2, stride=2
        )
        self.conv_decoder_3 = Block((features * 4) * 2, features * 4)
        self.upconv2 = nn.ConvTranspose2d(
            features * 4, features * 2, kernel_size=2, stride=2
        )
        self.conv_decoder_2 = Block((features * 2) * 2, features * 2)
        self.upconv1 = nn.ConvTranspose2d(
            features * 2, features, kernel_size=2, stride=2
        )
        self.decoder1 = Block(features * 2, features)

        self.conv = nn.Conv2d(
            in_channels=features, out_channels=out_channels, kernel_size=1
        )

    def forward(self, x):
        conv_encoder_1_1 = self.conv_encoder_1(x)
        conv_encoder_1_2 = nn.MaxPool2d(kernel_size=2, stride=2)(conv_encoder_

```

```
        conv_encoder_2_1 = self.conv_encoder_2(conv_encoder_1_2)
        conv_encoder_2_2 = nn.MaxPool2d(kernel_size=2, stride=2)(conv_encoder_

        conv_encoder_3_1 = self.conv_encoder_3(conv_encoder_2_2)
        conv_encoder_3_2 = nn.MaxPool2d(kernel_size=2, stride=2)(conv_encoder_

        conv_encoder_4_1 = self.conv_encoder_4(conv_encoder_3_2)
        conv_encoder_4_2 = nn.MaxPool2d(kernel_size=2, stride=2)(conv_encoder_

        bottleneck = self.bottleneck(conv_encoder_4_2)

        conv_decoder_4_1 = self.upconv4(bottleneck)
        conv_decoder_4_2 = torch.cat((conv_decoder_4_1, conv_encoder_4_1), dim
        conv_decoder_4_3 = self.conv_decoder_4(conv_decoder_4_2)

        conv_decoder_3_1 = self.upconv3(conv_decoder_4_3)
        conv_decoder_3_2 = torch.cat((conv_decoder_3_1, conv_encoder_3_1), dim
        conv_decoder_3_3 = self.conv_decoder_3(conv_decoder_3_2)

        conv_decoder_2_1 = self.upconv2(conv_decoder_3_3)
        conv_decoder_2_2 = torch.cat((conv_decoder_2_1, conv_encoder_2_1), dim
        conv_decoder_2_3 = self.conv_decoder_2(conv_decoder_2_2)

        conv_decoder_1_1 = self.upconv1(conv_decoder_2_3)
        conv_decoder_1_2 = torch.cat((conv_decoder_1_1, conv_encoder_1_1), dim
        conv_decoder_1_3 = self.decoder1(conv_decoder_1_2)

        return torch.sigmoid(self.conv(conv_decoder_1_3))
```

到这里，网络结构我们就搭建好了，然后我们来我看看损失函数。

损失函数：Dice Loss

这里我们来看一下语义分割中常用的损失函数，Dice Loss。

想要知道这个损失函数如何生成，你需要先了解一个语义分割的评价指标（但更常用的还是后面要讲的的 mIoU），它就是 Dice 系数，常用于计算两个集合的相似度，取值范围在 0-1 之间。

Dice 系数的公式如下。

$$Dice = \frac{2|P \cap G|}{|P| + |G|}$$

其中，$|P \cap G|$ 是集合 P 与集合 G 之间交集元素的个数，$|P|$ 和 $|G|$ 分别表示集合 P 和 G 的元素个数。分子的系数 2，这是为了抵消分母中 P 和 G 之间的共同元素。对语义分割任务而言，集合 P 就是预测值的 Mask，集合 G 就是真实值的 Mask。

根据 Dice 系数我们就能设计出一种损失函数，也就是 Dice Loss。它的计算公式非常简单，如下所示。

$$DiceLoss = 1 - \frac{2|P \cap G|}{|P| + |G|}$$

从公式中可以看出，当预测值的 Mask 与 GT 越相似，损失就越小；当预测值的 Mask 与 GT 差异度越大，损失就越大。

对于二分类问题，GT 只有 0 和 1 两个值。当我们直接使用模型输出的预测概率而不是使用阈值将它们转换为二值 Mask 时，这种损失函数就被称为 Soft Dice Loss。此时，$|P \cap G|$ 的值近似为 GT 与预测概率矩阵的点乘。

定义损失函数的代码如下。

复制代码

```
import torch.nn as nn

class DiceLoss(nn.Module):
    def __init__(self):
        super(DiceLoss, self).__init__()
        self.smooth = 1.0

    def forward(self, y_pred, y_true):
        assert y_pred.size() == y_true.size()
        y_pred = y_pred[:, 0].contiguous().view(-1)
        y_true = y_true[:, 0].contiguous().view(-1)
        intersection = (y_pred * y_true).sum()
        dsc = (2. * intersection + self.smooth) / (
            y_pred.sum() + y_true.sum() + self.smooth
        )
        return 1. - dsc
```

其中，self.smooth 是一个平滑值，这是为了防止分子和分母为 0 的情况。

训练流程

最后，我们将模型、损失函数和优化方法串起来，看下整体的训练流程，训练的代码如下。

复制代码

```
def main(args):
    makedirs(args)
    # 根据cuda可用情况选择使用cpu或gpu
    device = torch.device("cpu" if not torch.cuda.is_available() else args.dev
    # 加载训练数据
    loader_train = data_loaders(args)
    # 实例化UNet网络模型
    unet = UNet(in_channels=Dataset.in_channels, out_channels=Dataset.out_chan
    # 将模型送入gpu或cpu中
    unet.to(device)
    # 损失函数
    dsc_loss = DiceLoss()
    # 优化方法
    optimizer = optim.Adam(unet.parameters(), lr=args.lr)

    loss_train = []
    step = 0
    # 训练n个Epoch
    for epoch in tqdm(range(args.epochs), total=args.epochs):
        unet.train()
        for i, data in enumerate(loader_train):
            step += 1
            x, y_true = data
            x, y_true = x.to(device), y_true.to(device)
            y_pred = unet(x)
            optimizer.zero_grad()
            loss = dsc_loss(y_pred, y_true)
            loss_train.append(loss.item())
            loss.backward()
            optimizer.step()
            if (step + 1) % 10 == 0:
                print('Step ', step, 'Loss', np.mean(loss_train))
                loss_train = []
        torch.save(unet, args.weights + '/unet_epoch_{}.pth'.format(epoch))
```

需要注意的点，我都在注释中进行了说明，你可以自己看一看。其实就是我们一直说的模型训练的那几件事情：数据加载、构建网络以及迭代更新网络参数。

我用训练数据训练了若干个 Epoch，同时也保存了若干个模型，保存为 pth 格式。到这里就完成了模型训练的整个环节，我们可以使用保存好的模型进行预测，来看看分割效果如何。

模型预测

现在我们要用训练生成的模型来进行语义分割，看看结果是什么样子的。

模型预测的代码如下。

复制代码

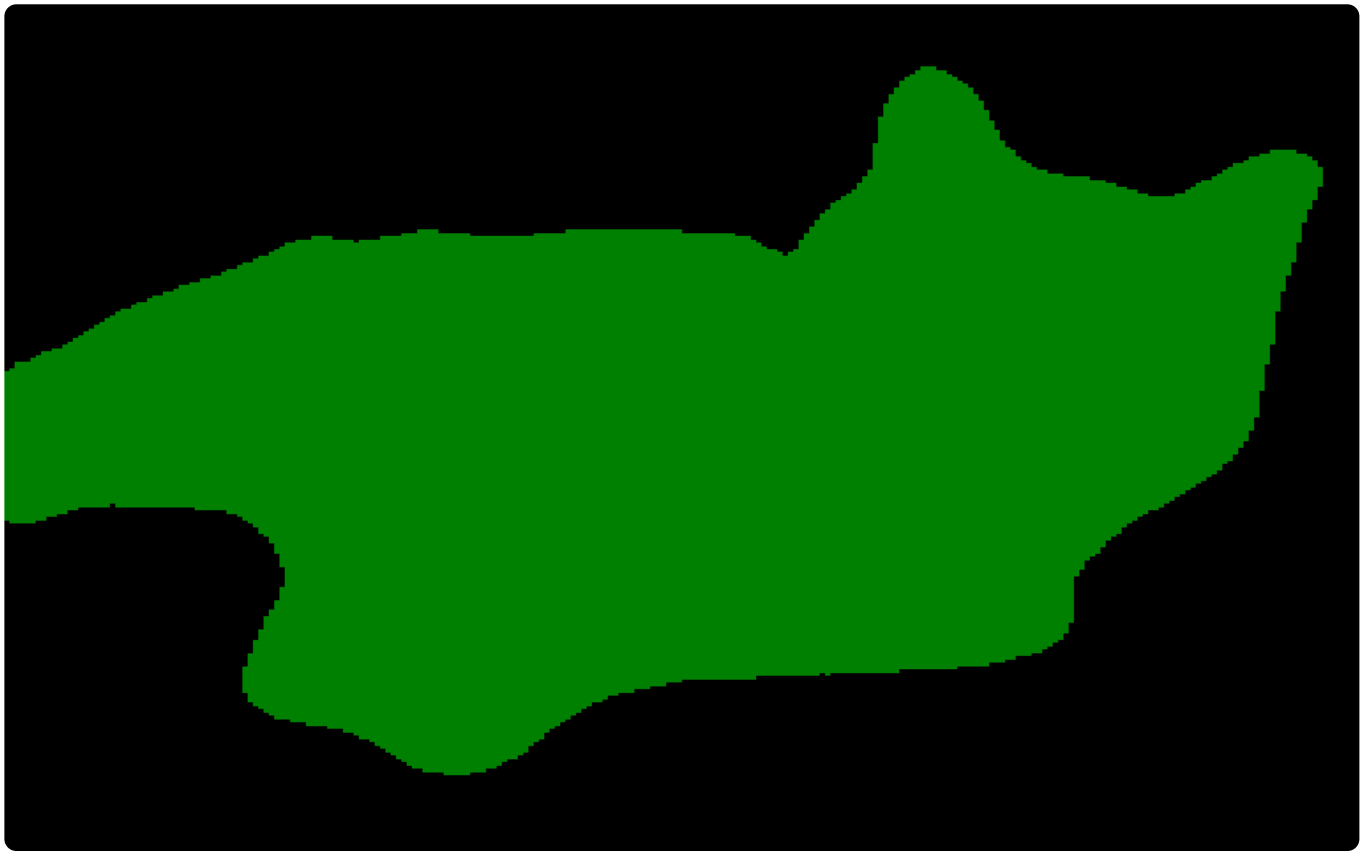
```
import torch
import numpy as np

from PIL import Image

img_size = (256, 256)
# 加载模型
unet = torch.load('./weights/unet_epoch_51.pth')
unet.eval()
# 加载并处理输入图片
ori_image = Image.open('data/JPEGImages/6.jpg')
im = np.asarray(ori_image.resize(img_size))
im = im / 255.
im = im.transpose(2, 0, 1)
im = im[np.newaxis, :, :]
im = im.astype('float32')
# 模型预测
output = unet(torch.from_numpy(im)).detach().numpy()
# 模型输出转化为Mask图片
output = np.squeeze(output)
output = np.where(output>0.5, 1, 0).astype(np.uint8)
mask = Image.fromarray(output, mode='P')
mask.putpalette([0,0,0, 0,128,0])
mask = mask.resize(ori_image.size)
mask.save('output.png')
```

这段代码也很好理解。首先，用 torch.load 函数加载模型。接着加载一张待分割的图片，并进行数据预处理。然后将处理好的数据送入模型中，得到预测值 output。最后将预测值转化为可视化的 Mask 图片进行保存。

输入图片也就是待分割的图片，如下左图所示。最终的输出，即可视化的 Mask 图片如下右图所示。



在将预测值转化为 Mask 图片的过程中，最终预测值的概率卡了 0.5 的阈值，超过阈值的像素点，在 output 矩阵中的值为 1，表示猫的区域，没有超过阈值的像素点，在 output 矩阵中的值为 0，表示背景区域。

为了将 output 矩阵输出为可视化的图像，我们使用 Image.fromarray 函数，将 Numpy 的 array 转化为 Image 格式，并将模式设置为“P”，即调色板模式。然后用 putpalette 函数来给 Image 对象上色。

其中，putpalette 函数的参数是一个列表：[0, 0, 0, 0, 128, 0]，列表前三个数表示值为 0 的像素的 RGB（[0, 0, 0]表示黑色），列表后三个数表示值为 1 的像素的 RGB（[0, 128, 0]表示绿色）。这样，我们保存的 Mask 图片，黑色部分即为背景区域，绿色部分即为猫的区域。

不过，这样分开的轮廓图，可能无法让我们很直观地看出语义分割的效果。所以我们将原图和 Mask 合成一张图片来看看效果。具体的代码如下。

复制代码

```
image = ori_image.convert('RGBA')
mask = mask.convert('RGBA')
# 合成
image_mask = Image.blend(image, mask, 0.3)
image_mask.save("output_mask.png")
```

首先，我们将原图 image 和 Mask 图片都转换为’ RGBA’ 带透明度的模式。然后使用 Image.blend 函数将两张图片合成一张图片，最后一个参数 0.3 表示 Mask 图片透明度为 30%，原图的透明度为 70%。

最终的结果如下图所示。

这样我们就可以直观地看出哪些地方预测得不准确了。

模型评估

在语义分割中，常用的评价指标是 mIoU。mIoU 全称为 mean Intersection over Union，即平均交并比。交并比是真实值和预测值的交集和并集之比。

真实值就是我们刚刚用 labelme 标注的 Mask，也是 Ground Truth（GT）。如下左图所示。

预测值就是模型预测出的 Mask，用 Prediction 表示。如后面右图所示。

交集是指真实值与预测值的交集，如下图黄色区域所示。并集是指真实值与预测值的并集，如下图蓝色区域所示。

通过上面几个图，我们很容易就能理解 mIoU 了。mIoU 的公式如下所示。

$$mIoU = \frac{1}{k}\sum_{i=1}^{k}\frac{P \cap G}{P \cup G}$$

其中，k 为所有类别数，在我们的例子中，只有“cat”一类，因此 k 为 1，我们通常不将背景计算到 mIoU 中；P 为预测值；G 是真实值。

小结

恭喜你，完成了今天的学习任务。这节课我们一起完成了一个图像分割项目的实践。

首先，我带你了解了图像分割的数据准备，需要使用 Labelme 工具为图像做标记。数据质量的好坏决定了最终模型的质量，所以你要对数据的标注好好把握。在使用 Labelme 标记完成之后，我们可以使用 label2voc.py 将 json 转换为 Mask。

之后我们学习了一种非常高效且实用的模型–UNet，并使用 PyTorch 实现了其网络结构。

然后，我为你讲解了图像分割的评估指标 mIoU 和损失函数 Dice Loss。

mIoU 的公式如下：

$$mIoU = \frac{1}{k}\sum_{i=1}^{k}\frac{P \cap G}{P \cup G}$$

mIoU 主要是从预测结果与 GT 的重合度这一角度，来衡量分割模型的好与坏的，它是图像分割中经常使用的评价指标。

最后，我们使用训练好的模型进行预测，并对分割结果进行了可视化绘制。相信通过之前学习的图像分类项目与今天学习的图像分割项目，对于图像处理，你会获得更深层次的理解。

每课一练

你可以根据今天的内容，自己动手试试建立一个图像分割模型，然后用一张图片来测一下效果如何。

欢迎你在留言区跟我交流讨论，也推荐你把今天的内容分享给更多同事、朋友，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 图像分割（上）：详解图像分割原理与图像分割模型

下一篇 21 | NLP基础（上）：详解自然语言处理原理与常用算法

更多学习推荐

精选留言 (5)

写留言

zhaobk
2021-11-29

老师您好。关于class Block(nn.Module): class UNet(nn.Module):这两个类，我有点不明白，为什么要这么实现？是根据论文来搭建的吗？如果需要实现其他算法，也是要根据相关论文来自己搭建这种结构吗？

作者回复: hello，你好。

你的疑问是为什么要单独写一个Block类吗？
不单独一写也可以，就是个人习惯。

如果疑问是为什么class UNet中是像文中那样堆叠卷积层的话。
那么回答是根据论文来搭建的。

如果需要实现其他算法，可以直接使用torchvision中封装好的模型，或者找找有没有已经实现好的开源代码，然后基于他们的代码做改进。
实际上，很少有项目是从0开始一点点写代码的。

共 2 条评论 >

vcjmhg

2021-11-27

老师您好，我自己有尝试复现deeplab v3+这个比较主流的语义分割网络，然后发现针对一些尺寸较大的目标其分割效果还是比较不错的，但是对小的目标分割效果很差，甚至好多时候都分割不到，请问针对这种小目标，除了在数据集上做处理外，还有哪些好的处理或者优化方法呢？

展开

作者回复: hello，小物体分割不好的一个原因是在backbone的特征提取过程中，小物体的信息已经被忽略了。
举个例子，假设输入是256x256的数据，经过多层的特征提取后特征图依次为128x128，64x64，32x32，16x16。有可能小物体的信息在32x32那里就消失了。
从这个角度出发，可以看看能不能调整网络最小的特征图的大小，来解决小物体分割不好的问题（比如Unet中，删除最后一次下采样）。另外，你也要看看小物体是不是resize到训练数据尺寸的时候已经没有了。

官
2021-11-26

感觉mIoU也可以用在语义分割模型里，不知道有没有这种可能

作者回复: 也可以，得改进一下。不过他和dice loss很像。

十里秋波不起雨
2021-11-26

收获颇丰

展开

作者回复: 谢谢认可，加油😊

hbuelgr
2021-11-26

收获很多，每天听听。

展开

作者回复: 感谢认可^^，希望与你一起学习进步。